Office de la propriét intellectuelle du Canada

Un organisme d'Industrie Canada

Canadian Intellectual Property Office

An Agency of Industry Canada

Bureau canadien des brevets

Certification

La présente atteste que les documents ci-joints, dont la liste figure ci-dessous, sont des copies authentiques des documents déposés au Bureau des brevets.

Canadian Patent Office

Certification

This is to certify that the documents attached hereto and identified below are true copies of the documents on file in the Patent Office.

Specification and Drawings, as originally filed, with Application for Patent Serial No: **2,322,613**, on October 6, 2000, by **IBM CANADA LIMITED - IBM CANADA LIMITÉE**, assignee of Aamer Sachedina, Michael J. Winer, Robert W. Lyle and Matthew A. Huras, for "Latch Mechanism for Concurrent Computing Environments".

Agent certificateur/Certifying Officer

November 27, 2000

Date

Canada

(CIPO 68)

OPIC CIPO

LATCH MECHANISM FOR CONCURRENT COMPUTING ENVIRONMENTS

ABSTRACT

A stealable latch mechanism for programming environments supporting concurrent tasks. The latch mechanism has a function for providing a latch to a requesting task, a function for a task holding a latch to release the held latch, a function for a task holding a latch to mark the held latch stealable, and a function for a task holding a latch marked stealable to mark the latch unstealable where the held latch is not marked stolen by another task. The function for providing a latch to a requesting task provides that the requesting task will receive on request a latch marked stealable and held by a holding task. Any resources associated with the stealable latch are placed in a consistent state prior to the requesting task accessing the resources.

LATCH MECHANISM FOR CONCURRENT COMPUTING ENVIRONMENTS

FIELD OF THE INVENTION

The present invention is directed to an improvement in computing systems and in particular to a latching mechanism for concurrent computer environments.

BACKGROUND OF THE INVENTION

Concurrent software systems establish primitives to allow concurrently executing tasks (threads or processes) to share common resources. Such primitives are often referred to as "latches".

It is known to use an exclusive latching mechansim to permit or deny access to a resource associated with a latch. If a first task holds a latch to a resource which permits only exclusive access, then a second task requiring the resource must wait until the holding task releases the latch before being able to access the latch.

More complex latches allow both shared and exclusive access to the resource. If a first task holds the latch to a resource in exclusive mode, then a second task requiring the resource must wait until the holding task releases the latch. Alternatively, if a first task holds the latch in share mode, then a second task will be granted access to the resource at the same time, if it requests the latch in share mode as well. If the second task requests the latch in exclusive mode, then it must wait until all tasks holding the latch in share mode release the latch.

Once a latch-holding task has used the latched resource, the resource may be in an inconsistent or unstable state which makes the resource unusable by other tasks. Therefore, before the holding task relinquishes the latch the holding task must carry out steps to return the resource to a consistent or stable state. This may be accomplished by the holding task executing a subroutine that will "clean up" the resource. The resource may then be used by other tasks.

From the description above, it can be seen that there is an overhead cost in executing computer code

containing concurrency primitives such as latches. This overhead comprises several hidden costs, including:

the cost of executing instructions required for the latch-holding task to give up the latch;

the cost of informing other tasks requesting the resource of the change of the latch's state when the latch is released;

the cost incurred in making the resource protected by the latch consistent so that it may be accessed by other tasks (the cost of executing the "clean-up subroutine").

In many systems, the same task repeatedly requests, and releases, the same latch before another task requests the latch. If the system overhead costs referred to above are high, the latching and unlatching steps will result in a decreased efficiency of the system resulting in the overall throughput of the system being reduced.

Certain prior art systems attempt to avoid the above inefficiency by permitting latch-holding task to continue to hold the latch rather than relinquishing it once the task is finished using the resource. In such a system, the latch-holding task must periodically poll the latch to see if there are waiting tasks which have requested it. However, this approach may starve a waiting task because although the resource is not being used, until the latch-holding task polls the latch, the latch is not made available. Further, whenever the latch-holding task polls the latch, CPU time is used..

It is therefore desirable to have a means of efficiently managing concurrency primitives such as latches with reduced system overhead by releasing latches only when necessary, while avoiding the need for polling the latch. Such efficient management would prevent starvation of waiting tasks while saving CPU time.

SUMMARY OF THE INVENTION

According to an aspect of the present invention there is provided an improved latch mechanism for

a programming environment for running concurrent tasks.

According to another aspect of the present invention, there is provided a stealable latch mechanism for programming environments supporting concurrent tasks, including means for providing a latch to a requesting task, means for a task holding a latch to release the held latch, means for a task holding a latch to mark the held latch stealable, and means for a task holding a latch marked stealable to mark the latch unstealable where the held latch is not marked stolen by another task, the means for providing a latch to a requesting task further including means for the requesting task to be provided, on request, with a latch marked stealable and held by a holding task, and means for placing any resources associated with the stealable latch in a consistent state prior to the requesting task accessing the resources.

According to another aspect of the present invention, there is provided the above stealable latch mechanism, further including a set of flags for marking a latch stealable, stolen or unstealable, by a task.

According to another aspect of the present invention, there is provided the above stealable latch mechanism, further including means for assigning a priority to a stealable latch and means for tasks to request a latch with a selectively determined priority, whereby a stealable latch is provided to a requesting task having a determined priority above a threshold value defined by the assigned priority for the stealable latch.

According to another aspect of the present invention, there is provided a method for providing access to a resource in a programming environment supporting concurrent tasks, including the steps of providing a latch to a requesting task where a task requests the latch to obtain access to the resource, the task holding the latch selectively releasing the held latch after accessing the resource, the task holding the latch selectively marking the held latch stealable, and the task holding the latch marked stealable marking the latch unstealable where the held latch is not marked stolen by another task, the step of providing a latch to a requesting task further including the step of providing to the

requesting task a latch marked stealable and held by a holding task, and placing the resources in a consistent state prior to the requesting task accessing the resource.

According to another aspect of the present invention, there is provided a computer program product for implementing a stealable latch mechanism for a programming environment supporting concurrent tasks, the computer program product including a computer usable medium having computer readable code means embodied in the medium, including

computer readable program code means providing a latch to a requesting task,

computer readable program code means for a task holding a latch to release the held latch,

computer readable program code means for a task holding a latch to mark the held latch stealable, and

computer readable program code means for a task holding a latch marked stealable to mark the latch unstealable where the held latch is not marked stolen by another task,

the computer readable program code means for providing a latch to a requesting task further including means for the requesting task to be provided, on request, with a latch marked stealable and held by a holding task, and means for placing any resources associated with the stealable latch in a consistent state prior to the requesting task accessing the resources.

According to another aspect of the present invention, there is provided a computer program product for use in a computer programming environment supporting concurrent tasks, the computer program product including a computer usable medium having computer readable program code means embodied in the medium for implementing an exclusive latch mechanism for associated resources, the computer program product including:

computer readable program code means to provide a latch to a requesting task whereby the task becomes a holding task and the latch becomes a held latch,

computer readable program code means for a holding task to release a held latch, whereby the task ceases to be a holding task,

computer readable program code means for a holding task to mark as stealable a latch held

by the holding task, the means to mark the held latch stealable including means to specify a flag and a cleanup subroutine, each being associated with the held latch and the holding task, and including means to set the flag to the value stealable, and

computer readable program code means for a holding task to mark a held latch unstealable, including:

means for determining the value of the associated flag and for setting the associated flag to the value unstealable and for indicating that the determined value is stealable, where the determined value of the flag is stealable,

means for determining the value of the associated flag and for indicating that the value of the associated flag is stolen where the determined value of the flag is stolen,

the computer readable program code means to provide a latch to a requesting task further including means for a requesting task to request a held latch, including

means for determining the value of the flag associated with the held latch and the holding task, and

means to provide the held latch to the requesting task where the flag value is stealable including,

means for executing the cleanup subroutine associated with the held latch and the holding task, and

means for setting the flag associated with the held latch and the holding task to the value stolen.

According to another aspect of the present invention, there is provided a computer program product for use in a computer programming environment supporting concurrent tasks, the computer program product including a computer usable medium having computer readable program code means embodied in the medium for implementing a full function latch mechanism for associated resources, the computer program product including:

computer readable program code means to provide a latch to a requesting task whereby the task becomes a holding task and the latch becomes a held latch,

computer readable program code means for a holding task to release a held latch, whereby the task ceases to be a holding task,

computer readable program code means for maintaining a set of flags, each flag in the set being associated with a latch and a holding task and each flag in the set selectively having one of the values stealable, unstealable or stolen,

computer readable program code means for a holding task to mark a latch held by the holding task stealable, the means to mark the held latch stealable including means to specify a cleanup subroutine associated with the held latch and the holding task, and means to assign the value stealable to the flag associated with the holding task and the latch, and

computer readable program code means for a holding task to mark a latch held by the holding task unstealable, including:

means for determining the value of the flag associated with the holding task, for setting the associated flag to the value unstealable and for indicating that the determined value is stealable, where the determined value of the flag is stealable,

means for determining the value of the flag associated with the holding task and for indicating that the value of the associated flag is stolen where the determined value of the flag is stolen,

the computer readable program code means to provide a latch to a requesting task further including means for a requesting task to request a held latch, the latch mode of the requesting task and the latch mode of the held latch not both being share mode, including

means for determining whether the value of each flag associated with the held latch is set to stealable, and

means to provide the held latch to the requesting task where each of the flag values is stealable including means for executing the cleanup subroutine associated with the held latch and the holding task, and means for setting each of the flag values to stolen.

According to another aspect of the present invention, there is provided the above computer program product, further including computer readable program code means for maintaining a count of the

number of tasks holding a specified latch, and computer readable program code means for maintaining a count, for a specified latch, of the number of flags in the set having the value stealable, and in which the means for determining whether the value of each flag associated with the held latch is set to stealable, includes means to compare the count of the number of tasks that have marked the latch stealable with the number of tasks that are holding that latch.

According to another aspect of the present invention, there is provided a method for the use of the above stealable latch mechanism, the method including the following steps

1. marking a latch requested by a candidate task as stealable wherever applicable,
2. maintaining statistics relating to the frequency with which the latch of the candidate task is stolen by other tasks, and
3. selectively requiring the candidate task to mark the latch as stealable, or releasing the latch, based on the statistics maintained relating to the frequency with which the latch is stolen.

Advantages of the present invention include a potential efficiency resulting from the ability for a task to reaccess a resource without the need to fully release and reacquire the latch for the resource and without the need to carry out a subroutine to put the resource in a consistent state.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 illustrates, in a block diagram format, example tasks and a cleanup subroutine used in the description of the latch mechanism of the preferred embodiment.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Figure 1 depicts tasks 10, 12, and cleanup subroutine 14. Tasks 10, 12 are tasks requiring the use of a resource or resources (not shown) that are subject to access control by a latch. Each of tasks 10, 12 include code that makes use of the latched resource. These are shown as code blocks 16, 18 in tasks 10, 12, respectively. In the example shown in Figure 1, code 16 may be repeatedly executed within task 10. In the description of the preferred embodiment, tasks are referred to as including

computer code and as executing on a computer system. It will be understood by those skilled in the art that the preferred embodiment is applicable to concurrent programming constructs such as threads and concurrent processes. The term task is used as a general term and is intended to include concurrent processes, threads, and the like.

The latch mechanism implemented in the preferred embodiment extends latches as used in the prior art. When the preferred embodiment latch mechanism is used in the prior art manner, a call to a latch() function is made by task 10 before code 16 is executed. When code 16 has completed execution, task 10 calls an unlatch() function. As is provided in prior art systems, in the latch mechanism of the preferred embodiment, task 10 must wait until the call to the latch() function returns the latch to task 10. When task 10 waits for the latch the task is placed on a list of waiters for the latch.

In this way, task 10 waits on the latch and then holds the exclusive latch while code 16 is executed. When execution of code 16 is completed, the unlatch() function is called and the latch is made available to other tasks. In a similar manner, this prior art use of the latch mechanism permits task 12 to bracket code 18 with calls to the latch() and unlatch() functions. If task 12 is waiting on the latch (and is therefore on the list of waiters maintained by the latch), the unlatch() function called by task 10 results in the latch waking up task 12. Task 12 then acquires the latch and is able to access the latched resource by executing code 18.

As is the case with prior art latch mechanisms, information regarding the state of the latch maintained, including its mode and the waiters on the latch. The mode indicates whether the latch is in SHARE or EXCLUSIVE mode. This information is used in the determination of whether a new requester is to be provided with the latch. A task requesting the latch will specify whether the latch is required in shared or exclusive mode. The latch also has a list of waiters associated with it which list contains information on the number of tasks which have queued to use the resource, and is used to determine whether any tasks need to be notified when the latch is unlatched by a holding task. In this way, the latch mechanism of the preferred embodiment is similar to that of prior art

concurrency primitives.

However, using the latch mechanism of the preferred embodiment also makes it possible for a task such as task 10, which executes such that code 16 is repeatedly executed (either repeatedly within task 10 or by repeated execution of task 10 itself) to mark a latch as STEALABLE, rather than relinquishing the latch entirely. With reference to the example of Figure 1, the initial execution of code 16 is preceded by a call to the latch() function, as in the prior art. However, at the termination of execution of code 16, the latch of the preferred embodiment permits task 10 carry out a soft release of the latch. This soft release, as described below, permits task 10 to avoid certain overhead costs associated with a call to the unlatch() function. This potentially makes the computer code more efficient where task 10 reacquires the latch with no intervening access to the resource by other tasks.

As in the prior art, data is associated with the preferred embodiment latch to specify a mode, a list of waiters, and a count of the number of waiters for the latch. In addition, the preferred embodiment latch mechanism provides that the latch may have a global flag and a subroutine associated with it for each holder of the latch (the latch may be held by multiple tasks if it is shareable). In the preferred embodiment, the latch maintains the address for the global flag and the subroutine for each task holding the latch.

The subroutine associated with the latch that is specified by a latch holder task is a subroutine that, when executed, will return the latched resource to a consistent or stable state. For the example of Figure 1, the code to clean up the resource after use by code 16 in task 10 is encapsulated in a subroutine that may be called by task 10 or by other tasks or threads. This is shown in Figure 1 as cleanup subroutine 14. The flag associated with the latch is used to indicate whether the latch is marked as STEALABLE, STOLEN or UNSTEALABLE, as is described below.

The programmer's interface for the latch of the preferred embodiment is implemented using two additional functions: mark_stealable() and mark_unstealable(). The first of these is used to carry out the soft release of the latch, referred to above. Thus where a task is holding the latch, but it is

expected that the task may seek to reaccess the latched resource imminently, the task may invoke the mark_stealable() function rather than the unlatch() function. The task will continue to hold the latch but it will be possible for other tasks to acquire or "steal" the latch.

When the mark_stealable() function is called, the task calling the function must provide a global flag and a cleanup subroutine (or pointers to them or addresses for them). The global flag for the function is set to the value STEALABLE by the mark_stealable function. While the latch is marked STEALABLE by the holding task, the holder cannot make use of the resource. Thus in the example of Figure 1, if task 10 calls the mark_stealable() function for the latch associated with code 16, the logic of task 10 must ensure that code 16 is not executed, except as set out below.

If the latch is in exclusive mode, after the latch is marked STEALABLE the holding task may seek to again use the resource, or a different task may seek to acquire the latch. Where the holding task seeks to access the resource, the task must first call the mark_unstealable() function. If the holder's global flag for the latch retains the value STEALABLE, then the mark_unstealable() function marks the flag UNSTEALABLE and the holding task may again access the resource.

In terms of the example of Figure 1, if task 10 holds the latch but has marked it STEALABLE and if task 10 again seeks to execute code 16, task 10 must first call the function mark_unstealable(). If mark_unstealable() returns with a value to indicate that the holder's global flag for that latch is STEALABLE then task 10 may execute code 16 to access the latched resource. Thus task 10 may avoid executing cleanup subroutine 14 between successive accesses to the resource made by code 18.

The second possible event referred to above occurs where a second task requests the exclusive latch before the holding task reaccesses the resource (by calling mark_stealable()). In this case, the second task (task 12 in the example of Figure 1), will seek to gain access to the latch in the usual manner, using the latch() function. In the preferred embodiment, the latch() function will provide the latch to the second task if it is available. The latch is available if it is not held by another task or if it is

held by another task in a mode compatible with the mode in which the second task is seeking the latch.

In addition, however, even where the latch is shown as being unavailable, a further check is carried out to determine if the latch being held by another task is marked STEALABLE. In the case of an exclusive latch, there will only be one global flag for the latch to be compared to determine if the latch is STEALABLE or not. If in such a case the latch is STEALABLE, then the latch() function will acquire the latch for the second task and will mark the latch STOLEN by the appropriate change to the value of the global flag. In addition, the cleanup subroutine associated with the latch by the holding task will be executed by the latch() function before accessing the resource. This ensures that the resource is returned to a stable state before access is permitted by another task.

In terms of the example of Figure 1, when task 10 calls mark_stealable() its flag for the latch is set to STEALABLE, and subroutine 14 is associated with the latch. When task 12 executes a latch() function (located prior to code 18), the flag will be marked STOLEN and cleanup subroutine 14 will be executed before execution of code 18.

Where the global flag associated with the latch for a holding task is set to STOLEN, and the holding task then seeks to access the latched resource by executing the mark_unstealable() function, the function returns the STOLEN value. The holding task must then call the latch() function to acquire the latch in the usual manner. In this case, the latch being marked stolen indicates that there has been an intervening use of the resource by another task and that the holding task's cleanup subroutine has been executed. Once the task acquires the latch by use of the latch() function, it is possible for the task to then use mark_stealable() to again make the latch stealable.

Where the latch is held in share mode, the latch() function contains logic such that a task calling the latch() function will only be permitted to steal the latch where all holders of the latch have marked the latch as stealable. In the preferred embodiment, the latch maintains not only the count of the number of current holders, but also the count of the number of tasks that marked the latch as

STEALABLE. These two numbers can therefore be compared in determining whether a latch in share mode is able to be stolen.

As will be apparent, the efficient use of the mark_stealable and mark_unstealable functions for the latch mechanism of the preferred embodiment by a task repeatedly using a resource will depend on an assessment of the likelihood that there will be interleaved requests for the resource by other tasks. Where the likelihood of such intervening requests for the resource is relatively low, or the cost of the clean up of the resource is relatively high, the use of the stealable latch mechanism is more advantageous. Where there are significant interleaved requests for the resource, or where there is minimal clean up of the resource required, the advantage to the stealable latch mechanism is lessened.

Therefore, stealable concurrency primitives are a viable option for reducing system overhead when:

the hidden cost of releasing the latch (making the resource consistent) is much higher than the combined cost to mark the latch stolen and to steal the latch; *and/or*

there is a low probability that the latch will be stolen; *and/or*

the cost of running the unlatch code is much greater than the cost to mark the latch stolen.

In the preferred embodiment, it is possible for a task to self-tune to determine whether there is a benefit to using the stealable aspect of the latch. A candidate task for use of the stealable feature of the latch initially makes use of the mark_stealable() function. The task maintains statistics regarding how often the mark_unstealable() function returns the STOLEN value. Where the proportion of returns with the STOLEN value is high, compared to the returns with the STEALABLE value, the task may revert to using the latch by calling latch() and unlatch(), only.

It will be understood by those skilled in the art that the preferred embodiment may be adapted to

other programming language environments in which a resource allocation method analogous to the latch concurrency primitive is provided.

Although a preferred embodiment of the present invention has been described here in detail, it will be appreciated by those skilled in the art that variations may be made thereto without departing from the spirit of the invention or the scope of the appended claims.

The embodiments of the invention in which an exclusive property or privilege are claimed are defined as follows:

1. A stealable latch mechanism for programming environments supporting concurrent tasks, comprising:

means for providing a latch to a requesting task,

means for a task holding a latch to release the held latch,

means for a task holding a latch to mark the held latch stealable, and

means for a task holding a latch marked stealable to mark the latch unstealable where the held latch is not marked stolen by another task,

the means for providing a latch to a requesting task further comprising means for the requesting task to be provided, on request, with a latch marked stealable and held by a holding task, and means for placing any resources associated with the stealable latch in a consistent state prior to the requesting task accessing the resources.

2. The stealable latch mechanism of claim 1, further comprising a set of flags for marking a latch stealable, stolen or unstealable, by a task.

3. The stealable latch mechanism of claim 1, further comprising means for assigning a priority to a stealable latch and means for tasks to request a latch with a selectively determined priority, whereby a stealable latch is provided to a requesting task having a determined priority above a threshold value defined by the assigned priority for the stealable latch.

4. A method for providing access to a resource in a programming environment supporting concurrent tasks, comprising the steps of:

providing a latch to a requesting task where a task requests the latch to obtain access to the resource,

the task holding the latch selectively releasing the held latch after accessing the resource,

the task holding the latch selectively marking the held latch stealable, and

the task holding the latch marked stealable marking the latch unstealable where the held latch is not marked stolen by another task,

the step of providing a latch to a requesting task further comprising the step of providing to the requesting task a latch marked stealable and held by a holding task, and placing the resources in a consistent state prior to the requesting task accessing the resource.

5. The method of claim 4, further comprising marking a set of flags for a latch stealable, stolen or unstealable as requested by a latch.

6. The method of claim 4, further comprising the step of assigning a priority to a stealable latch and the step of a task requesting a latch with a selectively determined priority, whereby a stealable latch is provided to a requesting task having a determined priority above a threshold value defined by the assigned priority for the stealable latch.

7. A computer program product for implementing a stealable latch mechanism for a programming environment supporting concurrent tasks, the computer program product comprising a computer usable medium having computer readable code means embodied in said medium, comprising

computer readable program code means providing a latch to a requesting task,

computer readable program code means for a task holding a latch to release the held latch,

computer readable program code means for a task holding a latch to mark the held latch

stealable, and

computer readable program code means for a task holding a latch marked stealable to mark the latch unstealable where the held latch is not marked stolen by another task,

the computer readable program code means for providing a latch to a requesting task further comprising means for the requesting task to be provided, on request, with a latch marked stealable and held by a holding task, and means for placing any resources associated with the stealable latch in a consistent state prior to the requesting task accessing the resources.

8. The computer program product of claim 7, further comprising computer readable program code means for implementing a set of flags for marking a latch stealable, stolen or unstealable, by a task.

9. The computer program product of claim 7, further comprising

computer readable program code means for assigning a priority to a stealable latch and

computer readable program code means for tasks to request a latch with a selectively determined priority,

whereby a stealable latch is provided to a requesting task having a determined priority above a threshold value defined by the assigned priority for the stealable latch.

10. A computer program product for use in a computer programming environment supporting concurrent tasks, said computer program product comprising a computer usable medium having computer readable program code means embodied in said medium for implementing an exclusive latch mechanism for associated resources, said computer program product comprising:

computer readable program code means to provide a latch to a requesting task whereby the task becomes a holding task and the latch becomes a held latch,

computer readable program code means for a holding task to release a held latch, whereby the task ceases to be a holding task,

computer readable program code means for a holding task to mark as stealable a latch held by the holding task, the means to mark the held latch stealable comprising means to specify a flag and a cleanup subroutine, each being associated with the held latch and the holding task, and comprising means to set the flag to the value stealable, and

computer readable program code means for a holding task to mark a held latch unstealable, comprising:

means for determining the value of the associated flag and for setting the associated flag to the value unstealable and for indicating that the determined value is stealable, where the determined value of the flag is stealable,

means for determining the value of the associated flag and for indicating that the value of the associated flag is stolen where the determined value of the flag is stolen,

the computer readable program code means to provide a latch to a requesting task further comprising means for a requesting task to request a held latch, comprising

means for determining the value of the flag associated with the held latch and the holding task, and

means to provide the held latch to the requesting task where the said flag value is stealable comprising,

means for executing the cleanup subroutine associated with the held latch and the holding task, and

means for setting the flag associated with the held latch and the holding task to the value stolen.

11. A computer program product for use in a computer programming environment supporting

concurrent tasks, said computer program product comprising a computer usable medium having computer readable program code means embodied in said medium for implementing a full function latch mechanism for associated resources, said computer program product comprising:

computer readable program code means to provide a latch to a requesting task whereby the task becomes a holding task and the latch becomes a held latch,

computer readable program code means for a holding task to release a held latch, whereby the task ceases to be a holding task,

computer readable program code means for maintaining a set of flags, each flag in the set being associated with a latch and a holding task and each flag in the set selectively having one of the values stealable, unstealable or stolen,

computer readable program code means for a holding task to mark a latch held by the holding task stealable, the means to mark the held latch stealable comprising means to specify a cleanup subroutine associated with the held latch and the holding task, and means to assign the value stealable to the flag associated with the holding task and the latch, and

computer readable program code means for a holding task to mark a latch held by the holding task unstealable, comprising

means for determining the value of the flag associated with the holding task, for setting the associated flag to the value unstealable and for indicating that the determined value is stealable, where the determined value of the flag is stealable,

means for determining the value of the flag associated with the holding task and for indicating that the value of the associated flag is stolen where the determined value of the flag is stolen,

the computer readable program code means to provide a latch to a requesting task further comprising means for a requesting task to request a held latch, the latch mode of the requesting task and the latch mode of the held latch not both being share mode, comprising

means for determining whether the value of each flag associated with the held latch

is set to stealable, and

means to provide the held latch to the requesting task where each of the said flag values is stealable comprising means for executing the cleanup subroutine associated with the held latch and the holding task, and means for setting each of the said flag values to stolen.

12. The computer program product of claim 11, further comprising

computer readable program code means for maintaining a count of the number of tasks holding a specified latch,

computer readable program code means for maintaining a count, for a specified latch, of the number of flags in the set having the value stealable,

and in which the means for determining whether the value of each flag associated with the held latch is set to stealable, comprises means to compare the count of the number of tasks that have marked the latch stealable with the number of tasks that are holding that latch.

13. A method for the use of the stealable latch mechanism claimed in claim 1, 2 or 3, the method comprising the following steps

a. marking a latch requested by a candidate task as stealable wherever applicable,

b. maintaining statistics relating to the frequency with which the latch of the candidate task is stolen by other tasks, and

c. selectively requiring the candidate task to mark the latch as stealable, or releasing the latch, based on the statistics maintained relating to the frequency with which the latch is stolen.